\[
\begin{align*}
\text{asinh} & \quad \rightarrow \text{asinh } a, \text{ acosh } a, \text{ or } \text{atanh } a, \text{ respectively.} \\
\text{acosh} & \quad \rightarrow \text{ Return } p^{1/n} = \cos a + i \sin a.
\end{align*}
\]

\[
\begin{align*}
\text{atanh} & \quad \rightarrow \text{ Return complex conjugate of } a.
\end{align*}
\]

\[
\begin{align*}
\text{max } \text{num}^+ & \quad \rightarrow \text{ Greatest or least, respectively, of } \text{num}s.
\end{align*}
\]

\[
\begin{align*}
\text{cis} & \quad \rightarrow e^{ia} = \cos a + i \sin a.
\end{align*}
\]

\[
\begin{align*}
\text{conjugate} & \quad \rightarrow \text{ Return complex conjugate of } a.
\end{align*}
\]

\[
\begin{align*}
\text{round} & \quad \rightarrow \text{ Return as integer or float, respectively, } n/d \text{ rounded, or rounded towards } -\infty, +\infty, \text{ or } 0, \text{ respectively; and remainder.}
\end{align*}
\]

\[
\begin{align*}
\text{mod} & \quad \rightarrow \text{ Same as } \text{floor} \text{ or } \text{truncate, respectively, but return remainder only.}
\end{align*}
\]

\[
\begin{align*}
\text{random} & \quad \rightarrow \text{ Return non-negative random number less than limit, and of the same type.}
\end{align*}
\]

\[
\begin{align*}
\text{make-random-state} & \quad \rightarrow \text{ Copy of random-state object state or of the current random state; or a randomly initialized fresh random state.}
\end{align*}
\]

\[
\begin{align*}
\text{random-state} & \quad \rightarrow \text{ Current random state.}
\end{align*}
\]

\[
\begin{align*}
\text{float-sign} & \quad \rightarrow \text{ num-b with num-a's sign.}
\end{align*}
\]

\[
\begin{align*}
\text{signum} & \quad \rightarrow \text{ Number of magnitude 1 representing sign or phase of } n.
\end{align*}
\]

\[
\begin{align*}
\text{numerator} & \quad \rightarrow \text{ Numerator or denominator, respectively, of rational's canonical form.}
\end{align*}
\]

\[
\begin{align*}
\text{realpart} & \quad \rightarrow \text{ Real part or imaginary part, respectively, of number.}
\end{align*}
\]

\[
\begin{align*}
\text{imagpart} & \quad \rightarrow \text{ Make a complex number.}
\end{align*}
\]

\[
\begin{align*}
\text{phase} & \quad \rightarrow \text{ Angle of num's polar representation.}
\end{align*}
\]

\[
\begin{align*}
\text{abs} & \quad \rightarrow \text{ Return } |n|.
\end{align*}
\]

\[
\begin{align*}
\text{rationalize} & \quad \rightarrow \text{ Convert real to rational. Assume complete/limited accuracy for real.}
\end{align*}
\]

\[
\begin{align*}
\text{float} & \quad \rightarrow \text{ Convert real into float with type of prototype.}
\end{align*}
\]

1.3 Logic Functions

Negative integers are used in two's complement representation.

\[
\begin{align*}
\text{boole operation int-a int-b} & \quad \rightarrow \text{ Return value of bitwise logical operation. Operations are}
\end{align*}
\]

\[
\begin{align*}
\text{boole-1} & \quad \rightarrow \text{ int-a.} \\
\text{boole-2} & \quad \rightarrow \text{ int-b.} \\
\text{boole-c1} & \quad \rightarrow \text{ ~int-a.} \\
\text{boole-c2} & \quad \rightarrow \text{ ~int-b.} \\
\text{boole-set} & \quad \rightarrow \text{ All bits set.} \\
\text{boole-clr} & \quad \rightarrow \text{ All bits zero.}
\end{align*}
\]
Contents

1 Numbers ........................................ 3 9.5 Control Flow .................. 19
1.1 Predicates ............................. 3 9.6 Iteration ......................... 21
1.2 Numeric Functions ........................ 3 9.7 Loop Facility ..................... 21
1.3 Logic Functions .......................... 4 14.4 Standard Packages ............... 42
1.4 Integer Functions ........................ 5 14.3 Symbols ......................... 43
1.5 Implementation-Dependent ................. 6 14.4 Standard Packages ............... 44
2 Characters ................................... 6
3 Strings ......................................... 7
4 Conses ....................................... 8
4.1 Predicates .................................. 8
4.2 Lists ..................................... 9
4.3 Association Lists .......................... 9
4.4 Trees .................................... 10
4.5 Sets ...................................... 10
5 Arrays ....................................... 10
5.1 Predicates .................................. 10
5.2 Array Functions ........................... 10
5.3 Vector Functions ........................... 11
6 Sequences ................................... 12
6.1 Sequence Predicates ....................... 12
6.2 Sequence Functions ....................... 12
7 Hash Tables ................................... 14
8 Structures .................................... 15
9 Control Structure ........................... 15
9.1 Predicates .................................. 15
9.2 Variables .................................. 16
9.3 Functions .................................. 17
9.4 Macros .................................... 18
16 External Environment ...................... 47

1.1 Predicates

(= number+)
(/= number+)
(> number+)
(>= number+)
(< number+)
(<= number+)

Return \( \sum a \) or \( \prod a \), respectively.

Return \( a - \sum b \) or \( a/\prod b \), respectively. Without any \( b s \),
return \( a - 1 \) or \( 1/a \), respectively.

Return \( a + 1 \) or \( a - 1 \), respectively.

Increment or decrement the value of \( place \) by \( delta \). Return
new value.

Return \( e^p \) or \( b^x \), respectively.

Return \( \log_b a \) or, without \( b, \ln a \).

Return \( \sqrt[n]{a} \) or \( a/n \) in complex numbers/natural numbers.

Least common multiple or greatest common denominator,
respectively, of \( \text{integers} \). \( \gcd \) returns \( 0 \).

\( \text{long-float} \) approximation of \( \pi \), Ludolph's number.

(\( \sin a \))
(\( \cos a \))
(\( \tan a \))
(\( \text{asin} a \))
(\( \text{acos} a \))
(\( \text{atan} a \))
(\( \text{atan2} b \))
(\( \text{sinh} a \))
(\( \cosh a \))
(\( \tanh a \))
4 Conses

4.1 Predicates

\((\text{cons} \ \text{foo} \ \text{bar})\)  \(\triangleright\) Return new cons \((\text{foo} , \ \text{bar})\).

\((\text{list} \ \text{foo})\)  \(\triangleright\) Return list of \(\text{foo}\).

\((\text{list*} \ \text{foo})\)  \(\triangleright\) Return list of \(\text{foo}\) with last \(\text{foo}\) becoming cdr of last cons. Return \(\text{foo}\) if only one \(\text{foo}\) given.

\((\text{make-list} \ \text{num} \ [\text{initial-element} \ \text{foo}])\)  \(\triangleright\) New list with \(\text{num}\) elements set to \(\text{foo}\).

\((\text{list-length} \ \text{list})\)  \(\triangleright\) Length of \(\text{list}\) is \(\text{NIL}\) for circular list.

\((\text{car} \ \text{list})\)  \(\triangleright\) Car of \(\text{list}\) or \(\text{NIL}\) if \(\text{list}\) is \(\text{NIL}\) \(\text{setf}\)able.

\((\text{cdr} \ \text{list})\)  \(\triangleright\) Cdr of \(\text{list}\) or \(\text{NIL}\) if \(\text{list}\) is \(\text{NIL}\) \(\text{setf}\)able.

\((\text{nthcdr} \ \text{n} \ \text{list})\)  \(\triangleright\) Return tail of \(\text{list}\) after calling \(\text{cdr} \ \text{n}\) times.

\((\{\text{first} \ \text{second} \ \text{third} \ \text{forth} \ \text{fifth} \ \text{sixth} \ldots \ \text{[nth]} \ \text{[tenth]}\} \ \text{list})\)  \(\triangleright\) Return \(\text{n}\)th element of \(\text{list}\) if \(\text{any}\) or \(\text{NIL}\) otherwise \(\text{setf}\)able.

\((\text{nth} \ \text{n} \ \text{list})\)  \(\triangleright\) Zero-indexed \(\text{n}\)th element of \(\text{list}\) \(\text{setf}\)able.

\((\text{cadr} \ \text{list})\)  \(\triangleright\) With \(X\) being one to four as and \(\text{ds}\) representing \(\text{cads}\) and \(\text{cdrs}\), e.g. \((\text{cadr} \ \text{bar})\) is equivalent to \((\text{car} \ (\text{cdr} \ \text{bar}))\) \(\text{setf}\)able.

\((\text{last} \ \text{list} \ [\text{num}\])\)  \(\triangleright\) Return list of last \(\text{num}\) conses of \(\text{list}\).

1.4 Integer Functions

\((\text{integer-length} \ \text{integer})\)  \(\triangleright\) Number of bits necessary to represent \(\text{integer}\).

\((\text{ldb-test} \ \text{byte-spec} \ \text{integer})\)  \(\triangleright\) Return \(\text{T}\) if any bit specified by \(\text{byte-spec}\) in \(\text{integer}\) is set.

\((\text{ash} \ \text{integer} \ \text{count})\)  \(\triangleright\) Return copy of \(\text{integer}\) arithmetically shifted by \(\text{count}\) adding zeros at the right, or, for \(\text{count} < 0\), shifted right discarding bits.

\((\text{ldb} \ \text{byte-spec} \ \text{integer})\)  \(\triangleright\) Extract \(\text{byte}\) denoted by \(\text{byte-spec}\) from \(\text{integer}\) \(\text{setf}\)able.

\((\text{deposit-field} \ \text{int-a} \ \text{byte-spec} \ \text{int-b})\)  \(\triangleright\) Return \(\text{int-b}\) with \(\text{bits}\) replaced by corresponding bits of \(\text{int-a}\), or by the low \(\text{(byte-size byte-spec) bits}\) of \(\text{int-a}\), respectively.

\((\text{mask-field} \ \text{byte-spec} \ \text{integer})\)  \(\triangleright\) Return copy of \(\text{integer}\) with all bits unset but those denoted by \(\text{byte-spec}\) \(\text{setf}\)able.

\((\text{byte} \ \text{size} \ \text{position})\)  \(\triangleright\) Byte specifier for a byte of \(\text{size}\) bits starting at a weight of \(\text{position}\).

\((\text{byte-size} \ \text{byte-spec})\)  \(\triangleright\) Size of \(\text{byte}\) specifier, respectively, of \(\text{byte-spec}\).
1.5 Implementation-Dependent

\( \text{radix}, \text{weight if string of length, a NIL number with name if any, or digits in that radix, or string, respectively.} \)

\( \text{character with 10 with 10 string, or an -lessp NIL character, re-} \)

\( \text{characterp foo} \)

\( \text{integer-decode-float n} \)

\( \text{scale-float n i} \)

\( \text{float-radix n} \)

\( \text{float-digits n} \)

\( \text{float-precision n} \)

\( \text{upgraded-complex-part-type foo} \)

\( \text{Type of most specialized complex number able to hold parts of type foo.} \)

2 Characters

The standard-char type comprises a-z, A-Z, 0-9, Newline, Space, and !?@#$%^&*()_+\-={}\[\]~":;\'\",\;\:\\-. If all characters, or none, respectively, are equal. Return \( \text{T} \) if character is uppercase, lowercase, or able to be in another case, respectively.

\( \text{digit-char-p character \text{radix}} \)

\( \text{Return its weight if character is a digit, or NIL otherwise.} \)

\( \text{char= character+} \)

\( \text{char/= character+} \)

\( \text{char= equal character+} \)

\( \text{char-\text{not-equal} character+} \)

\( \text{char\text{\textgreater} character+} \)

\( \text{char\textlessthan character+} \)

\( \text{char\textlessthan character+} \)

\( \text{Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.} \)

3 Strings

Strings can as well be manipulated by array and sequence functions; see pages 10 and 12.

\( \text{stringp foo} \)

\( \text{simple-string-p foo} \)

\( \text{Return T if foo is of indicated type.} \)

\( \text{string= string-equal} \)

\( \text{Return T if subsequences of foo and bar are equal.} \)

\( \text{make-string size \{\text{\textit{initial-element char}} \text{\textit{element-type type}}\}} \)

\( \text{Return string of length size.} \)

\( \text{string x} \)

\( \text{string\text{-capitalize}\ x} \)

\( \text{string\text{-upcase}\ x} \)

\( \text{string\text{-downcase}\ x} \)

\( \text{string\text{-capitalize}\ x} \)

\( \text{string\text{-upcase}\ x} \)

\( \text{string\text{-downcase}\ x} \)

\( \text{Return string into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.} \)

\( \text{string\text{-trim}\ x} \)

\( \text{Return string with all characters in sequence char-bag removed from both ends, from the beginning, or from the end, respectively.} \)
6 Sequences

6.1 Sequence Predicates

{every (not every)} test sequence+  
▷ Return `NIL` or `T`, respectively, as soon as `test` on any set of corresponding elements of `sequence`s returns `NIL`.

{some (not some)} test sequence+  
▷ Return value of `test` or `NIL`, respectively, as soon as `test` on any set of corresponding elements of `sequence`s returns non-`NIL`.

{mismatch sequence-a sequence-b}  
▷ Return position in `sequence-a` where `sequence-a` and `sequence-b` begin to mismatch. Return `NIL` if they match entirely.

6.2 Sequence Functions

{make-sequence sequence-type size [initial-element foo]}  
▷ Make sequence of `sequence-type` with `size` elements.

{concatenate type sequence*}  
▷ Return concatenated sequence of type.

{merge type sequence-a sequence-b test [key function test]}  
▷ Return interleaved sequence of `type`. Merged sequence will be sorted if both `sequence-a` and `sequence-b` are sorted.

{fill sequence foo}  
▷ Return `sequence` after setting elements between `start` and `end` to `foo`.

{length sequence}  
▷ Return length of `sequence` (being value of fill pointer if applicable).

{count foo sequence}  
▷ Return number of elements in `sequence` which match `foo`.

{count-if (not count-if-not)} test sequence  
▷ Return number of elements in `sequence` which satisfy `test`.

{elt sequence index}  
▷ Return element of `sequence` pointed to by zero-indexed `index`. `setf`able.

{subseq sequence start [end]}  
▷ Return subsequence of `sequence` between `start` and `end`. `setf`able.

{sort (stable-sort) sequence test [key function]}  
▷ Return `sequence` sorted. Order of elements considered equal is not guaranteed/retained, respectively.

{reverse sequence}  
▷ Return `sequence` in reverse order.

{reverse sequence}  
▷ Return `sequence` in reverse order.

4.3 Association Lists

{pairwise keys value alist}  
▷ Prepend to `alist` an association list made from `keys` and `values`.

{acons key value alist}  
▷ Return `alist` with a `(key, value)` pair added.

{assoc foo alist}  
▷ Return (possibly empty) sequence of elements tested `foo`.

{assoc-if-not}  
▷ Return first cons whose car, or cdr, respectively, satisfies `test`.  

{copy-alist alist}  
▷ Return copy of `alist`.  

{revappend list foo}  
▷ Return concatenated list after reversing order in `list`.

{append [proper-list* foo]}  
▷ Return concatenated list or, with only one argument, `foo` can be of any type.

{copy-list list}  
▷ Return copy of `list` with shared elements.
4.4 Trees

(defun tree-equal foo bar
  "Return T if trees foo and bar have same shape and leaves satisfying Test."
  (cond ((test foo) (test bar)) nil))

(defun subst new-old-tree
  "Make copy of tree with each subtree or leaf matching old replaced by new."
  (cond ((test new-old-tree) (key function)) nil))

(defun subst-if-not new-old-tree
  "Make copy of tree with each subtree or leaf satisfying test replaced by new."
  (cond ((test new-old-tree) (key function)) nil))

(defun sublis association-list-tree
  "Get list containing the lengths of tree dimension of tree array."
  (cond ((test association-list-tree) (key function)) nil))

(defun copy-tree
  "Copy of tree with same shape and leaves."
  (cond ((test tree) (key function)) nil))

4.5 Sets

(defun intersection
  "Return a \(\cap\) b, a \(\setminus\) b, a \(\cup\) b, or a \(\triangle\) b, respectively, of lists a and b."
  (cond ((test a) (test b)) nil))

(defun set-exclusive-or
  "Return a \(\oplus\) b, a \(\ominus\) b, and a \(\ominus\) b, respectively, of lists a and b."
  (cond ((test a) (test b)) nil))

(defun union
  "Return a \(\cup\) b, a \(\ominus\) b, and a \(\ominus\) b, respectively, of lists a and b."
  (cond ((test a) (test b)) nil))

(defun set-exclusive-or
  "Return a \(\ominus\) b, a \(\ominus\) b, and a \(\ominus\) b, respectively, of lists a and b."
  (cond ((test a) (test b)) nil))

5 Arrays

5.1 Predicates

(defun array-p foo)
  "Return T if foo is of indicated type."

(defun vectorp foo)
  "Return T if foo is of indicated type."

(defun simple-vector-p foo)
  "Return T if foo is of indicated type."

(defun bit-vector-p foo)
  "Return T if foo is of indicated type."

(defun adjustable-array-p array
  "Return T if array is adjustable/has a fill pointer, respectively."

(defun array-has-fill-pointer-p array)
  "Return T if array is adjustable/has a fill pointer, respectively."

(defun array-in-bounds-p array [subscripts])
  "Return T if subscripts are in array’s bounds."

5.2 Array Functions

(defun make-array
dimension-sizes :adjustable boolean)
  "Return array dimension-sizes."

(defun adjust-array
dimension-sizes
  "element-type type
  :fill-pointer [num] boolean
  :initial-element object
  :initial-contents tree-or-array
  :displaced-to array [displaced-index-offset]
  "Return fresh, or readjust, respectively, vector or array."

(defun aref array [subscripts])
  "Return array element pointed to by subscripts. setfable."

(defun row-major-aref array i)
  "Return i\(th\) element of array in row-major order. setfable."
9.2 Variables

(defconstant foo [doc])
▷ Assign value of form to global constant/dynamic variable foo.

(defvar foo [form [doc]])
▷ Unless bound already, assign value of form to dynamic variable foo.

(setf place form)^
▷ Set places to primary values of forms. Return values of last form/NIL; work sequentially/in parallel, respectively.

(setq symbol form)^
▷ Set symbols to primary values of forms. Return value of last form/NIL; work sequentially/in parallel, respectively.

(set symbol foo)
▷ Set symbol’s value cell to foo. Deprecated.

(multiple-value-setq vars form)
▷ Set elements of vars to the values of form. Return form’s primary value.

(shift place + foo)
▷ Store value of foo in rightmost place shifting values of places left, returning first place.

(rotate place ^)
▷ Rotate values of places left, old first becoming new last place’s value. Return NIL.

(makunbound foo)
▷ Delete special variable foo if any.

(get symbol key [default])

(getf place key [default])
▷ First entry key from property list stored in symbol/in place, respectively, or default if there is no key. setable.

(get-properties property-list keys)
▷ Return key and value of first entry from property-list matching a key from keys, and tail of property-list starting with that key. Return NIL, NIL, and NIL if there was no matching key in property-list.

(remove symbol key)

(remf place key)
▷ Remove first entry key from property list stored in symbol/in place, respectively. Return T if key was there, or NIL otherwise.

(prov symbols values form)
▷ Evaluate forms with locally established dynamic bindings of symbols to values or NIL. Return values of forms.

(let ^)
▷ Evaluate forms with names lexically bound (in parallel or sequentially, respectively) to values. Return values of forms.

(multiple-value-bind (var *) values-form (declare dec ^)
body-form)
▷ Evaluate body-forms with vars lexically bound to the return values of values-form. Return values of body-forms.
7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see loop, page 21.

Key-value storage similar to hash tables can as well be achieved using association lists and property lists; see pages 9 and 16.

(defun hash-table-p (foo)    ; Return T if foo is of type hash-table.
  (not (or (type-of foo) (type-of *hash-table*))))

(defun make-hash-table (size num)              ; Make a hash table.
  (make-hash-table :initial-size size :initial-value num))

(defun gethash key hash-table (default)        ; Return object with key if any or default otherwise; and T if found, NIL otherwise. setf table.
  (gethash key hash-table default))

(defun remhash key hash-table)                 ; Remove from hash-table entry with key and return T if it existed. Return NIL otherwise.
  (remhash key hash-table))

(defun clhash hash-table)                      ; Empty hash-table.
  (empty hash-table))

(defun maphash function hash-table)            ; Iterate over hash-table calling function on key and value. Return NIL.
  (maphash function hash-table))

(defun with-hash-table-iterator (foo hash-table) (declare (decl-type) format)) ; Return values of forms. In forms, invocations of (foo) return: T if an entry is returned; its key; its value.
  (with-hash-table-iterator foo hash-table)

(defun hash-table-test hash-table)             ; Test function used in hash-table.
  (hash-table-test hash-table)

(defun hash-table-size hash-table)             ; Current size, rehash-size, or rehash-threshold, respectively, as used in make-hash-table.
  (hash-table-size hash-table)

(defun hash-table-rehash-threshold hash-table) ; Current size, rehash-size, or rehash-threshold, respectively, as used in make-hash-table.
  (hash-table-rehash-threshold hash-table)

(defun sxhash foo)                             ; Hash code unique for any argument equal foo.
  (sxhash foo)
Return the values of the first foo* one of whose keys is eql test. Return values of bars if there is no matching key.

Return the values of the first foo* one of whose keys is eql test. Signal non-correctable/correctable type-error if there is no matching key.

(m and form*)

Evaluate forms from left to right. Immediately return NIL if one form’s value is NIL. Return values of last form otherwise.

(mor form*)

Evaluate forms from left to right. Immediately return primary value of first non-NIL-evaluating form, or all values if last form is reached. Return NIL if no form returns T.

(progn form*)

Evaluate forms sequentially. Return values of last form.

(multiple-value-prog1 form-r form*)

(multiple-value-prog2 form-r form*)

Evaluate forms in order. Return values/primary value, respectively, of form-r.

Evaluate tagbody-like body with names lexically bound (in parallel or sequentially, respectively) to values. Return NIL or explicitly returned values. Implicitly, the whole form is a block named NIL.

Evaluate protected and then, no matter how control leaves protected, cleansups. Return values of protected.

Evaluate forms in a lexical environment, and return their values unless interrupted by return-from.

Have nearest enclosing block named foo/named NIL, respectively, return with values of result.

Evaluate forms in a lexical environment. tags (symbols or integers) have lexical scope and dynamic extent, and are targets for go. Return NIL.

Within the innermost possible enclosing tagbody, jump to a tag if eql tag.

Evaluate forms and return their values unless interrupted by throw.

Have the nearest dynamically enclosing catch with a tag req tag return with the values of form.

Wait n seconds; return NIL.

(=destructuring-bind destruct-lambda (declare decl*)* form*)

Evaluate forms with variables from tree destruct-lambda bound to corresponding elements of tree bar, and return their values. destruct-lambda resembles macro-lambda (section 9.4), but without any &environment clause.

9.3 Functions

Below, ordinary lambda list (ord-* ) has the form

(%, defun foo (ord-* )

(lambda (ord-* )

(flet ((foo (ord-* )

(labels ((local-decl*)

(multiple-value-call function form*)

(multiple-value-list form)

(mth-value n form)

(complement function)

(constantly foo)

(identity foo)
9.4 Macros

Below, macro lambda list \((\text{macro-}\lambda^*)\) has the form of either

\[
\begin{align*}
&\text{\&whole \ var \ } \ [E] \ \text{\{var \ (macro-\lambda^*)\} } \ [E] \\
&\text{\&optional \ } \ \text{\{var \ (macro-\lambda^*)\} } \ \text{\{init \ (supplied-p)\} } \ [E] \\
&\text{\&key \ } \ \text{\{var \ (rest-var)\} } \ \text{\{key \ (macro-\lambda^*)\} } \ [E] \\
&\text{\&allow-other-keys} \ \text{\&aux \ var \ (var \ (init\#)) \ } \ [E] \\
&\text{\&body \ (macro-\lambda^*)} \ [E] \\
\end{align*}
\]

One toplevel \([E]\) may be replaced by \&environment \ var. supplied-p is \(T\) if there is a corresponding argument. \text{init} forms can refer to any init and supplied-p to their left.

\((\text{\&define-macro}) \ \text{\{foo \ (setf \ foo)\} \ \text{(macro-\lambda^*)}}\)

\text{\&define-macro} \ foo \ which on evaluation as \(\text{(foo tree)}\) applies expanded forms to arguments from \(\text{tree}\), which corresponds to tree-shaped macro-\(\lambda^s.\) forms are enclosed in an implicit block named \text{foo}.

\((\text{\&define-sym-macro}) \ \text{foo \ form})\)

\text{\&define-sym-macro} \ foo \ which on evaluation evaluates expanded form.

\((\text{\&macrolet}) \ \text{\{foo \ (macro-\lambda^*)\} \ \text{(declare local-decl*)} \ \text{form^b}}\)

\text{\&macrolet} \ foo \ which on evaluation as \(\text{foo tree}\) applies forms to arguments from \(\text{tree}\), which corresponds to tree-shaped macro-\(\lambda^s.\) forms are enclosed in an implicit block of the same name.

\((\text{\&symbol-macrolet}) \ \text{\{foo \ expansion-form\)} \ \text{(declare decl*)} \ \text{form^b}}\)

\text{\&symbol-macrolet} \ foo \ which on evaluation evaluates expanded form.

\((\text{\&defsetf}) \ \text{\{function \ lambda \ expression \ function\) \ \text{\{declare \ decl*\}} \ \text{form^b}}\)

\text{\&defsetf} \ lambda list \(\text{(setf-\lambda^*) \ (s-var*)} \ \text{(declare \ decl*)} \ \text{form^b}}\)

where \text{defsetf} lambda list \(\text{(setf-\lambda^*)} \) has the form

\[
\begin{align*}
&\text{\{var* \ \&optional \ var \ \text{init\# \ (supplied-p)}\} } \ [E] \\
&\text{\&key \ \{var \ (key \ var) \ \text{init\# \ (supplied-p)}\} } \\
&\text{\&allow-other-keys} \ \text{\&environment \ var} \\
&\text{\&body \ \text{var}} \\
\end{align*}
\]
\((\text{slot-boundp} \text{ instance slot}) \rightarrow \) \(T\) if slot in instance is bound.

\((\text{ofclass} \text{ foo (superclass* \text{standard-object})}) \rightarrow \) \([\text{slot} \text{ reader reader*}]\) \([\text{writer} \text{ (setf writer*)}]\)

\((\text{class-of} \text{ foo}) \rightarrow \) \(\text{class}\) named \(\text{foo}\) is a direct instance of.

\((\text{class-name} \text{ class}) \rightarrow \) Get/set name of class.

\((\text{ofclass} \text{ foo}) \rightarrow \) \(\text{class}\) \text{foo} is a direct instance of.

\((\text{change-class} \text{ instance new-class (\text{superclass* \text{standard-object})})}) \rightarrow \) Change \(\text{class}\) of \text{instance} to \text{new-class}. Retain the status of any slots that are common between \text{instance’s} original class and \text{new-class}. Initialize any newly added slots with the values of the corresponding \text{initargs} if any, or with the values of their \text{:initform} forms if not.

\((\text{make-classes-obsolete} \text{ class}) \rightarrow \) Update all existing instances of \text{class} using \text{make-instance-for-redefined-class}.

\((\text{initialize-instance} \text{ instance}) \rightarrow \) \(\text{class}\) of \text{instance} and \text{class} of \text{instance} only is \text{class} if any.

\((\text{make-classes-obsolete} \text{ class}) \rightarrow \) Update all existing instances of \text{class} using \text{make-instance-for-redefined-class}.

\((\text{initialize-instance} \text{ instance}) \rightarrow \) \(\text{class}\) of \text{instance} and \text{class} of \text{instance} only is \text{class} if any.

\((\text{make-classes-obsolete} \text{ class}) \rightarrow \) Update all existing instances of \text{class} using \text{make-instance-for-redefined-class}.

\((\text{class-name} \text{ class}) \rightarrow \) Get/set name of class.

\((\text{ofclass} \text{ foo}) \rightarrow \) \(\text{class}\) \text{foo} is a direct instance of.

\((\text{change-class} \text{ instance new-class (\text{superclass* \text{standard-object})})}) \rightarrow \) Change \(\text{class}\) of \text{instance} to \text{new-class}. Retain the status of any slots that are common between \text{instance’s} original class and \text{new-class}. Initialize any newly added slots with the values of the corresponding \text{initargs} if any, or with the values of their \text{:initform} forms if not.

\((\text{make-classes-obsolete} \text{ class}) \rightarrow \) Update all existing instances of \text{class} using \text{make-instance-for-redefined-class}.

\((\text{initialize-instance} \text{ instance}) \rightarrow \) \(\text{class}\) of \text{instance} and \text{class} of \text{instance} only is \text{class} if any.

\((\text{make-classes-obsolete} \text{ class}) \rightarrow \) Update all existing instances of \text{class} using \text{make-instance-for-redefined-class}.

\((\text{class-name} \text{ class}) \rightarrow \) Get/set name of class.

\((\text{ofclass} \text{ foo}) \rightarrow \) \(\text{class}\) \text{foo} is a direct instance of.

\((\text{change-class} \text{ instance new-class (\text{superclass* \text{standard-object})})}) \rightarrow \) Change \(\text{class}\) of \text{instance} to \text{new-class}. Retain the status of any slots that are common between \text{instance’s} original class and \text{new-class}. Initialize any newly added slots with the values of the corresponding \text{initargs} if any, or with the values of their \text{:initform} forms if not.

\((\text{make-classes-obsolete} \text{ class}) \rightarrow \) Update all existing instances of \text{class} using \text{make-instance-for-redefined-class}.

\((\text{initialize-instance} \text{ instance}) \rightarrow \) \(\text{class}\) of \text{instance} and \text{class} of \text{instance} only is \text{class} if any.

\((\text{make-classes-obsolete} \text{ class}) \rightarrow \) Update all existing instances of \text{class} using \text{make-instance-for-redefined-class}.

\((\text{class-name} \text{ class}) \rightarrow \) Get/set name of class.

\((\text{ofclass} \text{ foo}) \rightarrow \) \(\text{class}\) \text{foo} is a direct instance of.
(do (form+)   
  (do (form+) 
    (do (form+)) 
  ) 
)

(return (form))  
  ▶ Return immediately, skipping any finally parts, with values of form or it.

(collect (collecting) (form) [[into list] [type]])  
  ▶ Collect values of form or it into list. If no list is given, collect into an anonymous list which is returned after termination.

(append (appending) (nconc (nconcing)) (form) [[into list] [type]])  
  ▶ Concatenate values of form or it, which should be lists, into list by the means of append or nconc, respectively. If no list is given, collect into an anonymous list which is returned after termination.

(count (counting) (form) [[into n] [type]])  
  ▶ Count the number of times the value of form or of it is T. If no n is given, count into an anonymous variable which is returned after termination.

(sum (summing) (form) [[into n] [type]])  
  ▶ Calculate the sum of the primary values of form or of it. If no sum is given, sum into an anonymous variable which is returned after termination.

(maximize (maximizing) (minimize (minimizing) (form) [[into max-min] [type]])  
  ▶ Determine the maximum or minimum, respectively, of the primary values of form or of it. If no max-min is given, use an anonymous variable which is returned after termination.

(initially (finally) (form+)  
  ▶ Evaluate forms before begin, or after end, respectively, of iterations.

(repeat nmax)  
  ▶ Terminate loop after nmax iterations; nmax is evaluated once.

(while (until) test)  
  ▶ Continue iteration until test returns NIL or T, respectively.

(always (never) test)  
  ▶ Terminate loop returning NIL and skipping any finally parts as soon as test is NIL or T, respectively. Otherwise continue loop with its default return value set to T.

(thereis test)  
  ▶ Terminate loop when test is T and return value of test, skipping any finally parts. Otherwise continue loop with its default return value set to NIL.

(loop-finish)  
  ▶ Terminate loop immediately executing any finally clauses and returning any accumulated results.

10 CLOS

10.1 Classes

(slot-exists-p foo bar)  
  ▶ T if foo has a slot bar.
(make-condition condition-type \{[:initarg-name value]\}^*)
   ▷ Return new instance of condition-type.

(signal \{condition condition-type \{[:initarg-name value]\}^*\})
   ▷ Unless handled, signal as condition, warning or error, respectively, condition or a new instance of condition-type or with format control and args (see page 36), simple-condition, simple-warning, or simple-error, respectively. From signal and warn, return NIL.

(error \{continue-control condition-concrete-arg\}^*)
   ▷ Unless handled, signal as correctable error condition or a new instance of condition-type or, with format control and args (see page 36), error. In the debugger, use format arguments continue-control and continue-args to tag the continue option. Return NIL.

(\{ignore-errors form\}^*)
   ▷ Return values of forms or, in case of errors, NIL and the condition.

:invoke-debugger \{condition\}
   ▷ Invoke debugger with condition.

(\{assert text \{\}(place\}^*)
   ▷ If test, which may depend on places, returns nil, signal as correctable error condition or a new instance of condition-type or, with format control and args (see page 36), error. When using the debugger’s continue option, places can be altered before re-evaluation of test. Return NIL.

(handler-case \{\}^*)
   ▷ Return values of forms after evaluating them with condition-types dynamically bound to their respective handler-functions of argument condition.

(with-simple-restart \{\}^*)
   ▷ Return values of forms unless restart is called during their evaluation. In this case, describe restart using format control and args (see page 36) and return NIL and T.

(restart-case \{\}^*)
   ▷ Return values of form or, if during evaluation of form one of the dynamically established restarts is called, the values of its restart-forms. A restart is visible under condition if (funcall #'test-fuction-condition) returns T. If presented in the debugger, restarts are described by string by #\{report-function\} and by #\{report-function\} of a stream. A restart can be called by (invoke-restart restart arg), where args match ord-\*λ, or by (invoke-restart-interactively-restart) where a list of the respective args is supplied by #\{arg-function\}. See page 17 for ord-\*λ.

allocate-instance class \{[:initarg-name value]\}^* other-keyarg^
   ▷ Return uninitialized instance of class. Called by make-instance.

(shared-initialize instance \{\}^*)
   ▷ Fill the initarg-slots of instance with the corresponding values, and fill those initarg-slots that are not initarg-slots with the values of their initarg forms.

(slot-missing \{\}^*)
   ▷ Called on attempted access to non-existing or unbound slot. Default methods signal error/unbound-slot, respectively. Not to be called by user.

10.2 Generic Functions

(next-method-p)
   ▷ T if enclosing method has a next method.

(defgeneric \{\}^*)
   ▷ Define or modify generic function \{\}. Remove any methods previously defined by defgeneric. \{\} and the lambda parameters required-var and var* must be compatible with existing methods. defmethod-args resemble those of defmethod. For \{\} see section 10.3.

(\{\}^*)
   ▷ Define or modify generic function \{\}. \{\} and lambda-list must be compatible with pre-existing generic function or with existing methods, respectively. Changes to method-class do not propagate to existing methods. For \{\} see section 10.3.

(defmethod \{\}^*)
   ▷ Before after around qualifier^*
Define new method for generic function foo, spec-vars specialize to either being of class or being eql bar, respectively. On invocation, vars and spec-vars of the new method act like parameters of a function with body form*. Forms are enclosed in an implicit block foo. Applicable qualifiers depend on the method-combination type; see section 10.3.

\{\text{add-method} \}
generic-function method
\> Add (if necessary) or remove (if any) method to/from generic-function.

\{\text{find-method} generic-function qualifiers specializers \{error\}\}
\> Return suitable method, or signal error.

\{\text{compute-applicable-methods} generic-function args\}
\> List of methods suitable for args, most specific first.

\{\text{call-next-method} args \}
\> From within a method, call next method with args; return its values.

\{\text{no-applicable-method} generic-function any\}
\> Called on invocation of generic-function on args if there is no applicable method. Default method signals error. Not to be called by user.

\{\text{invalid-method-error} method \}
\{\text{method-combination-error} \}
\> Signal error on applicable method with invalid qualifiers, or on method combination. For control and args see format, page 36.

\{\text{no-next-method} generic-function method args\}
\> Called on invocation of call-next-method when there is no next method. Default method signals error. Not to be called by user.

\{\text{function-keywords} method\}
\> Return list of keyword parameters of method and T if other keys are allowed.

\{\text{method-qualifiers} method\} \> List of qualifiers of method.

10.3 Method Combination Types

\text{standard}
\> Evaluate most specific \text{around} method supplying the values of the generic function. From within this method, \text{call-next-method} can call less specific \text{around} methods if there are any. If not, or if there are no \text{around} methods at all, call all \text{before} methods, most specific first, and the most specific primary method which supplies the values of the calling \text{call-next-method} if any, or of the generic function; and which can call less specific primary methods via \text{call-next-method}. After its return, call all \text{after} methods, least specific first.

and/or append list [\text{block} prog] max/min \> Simple built-in method combination types; have the same usage as the \text{c-types} defined by the short form of \text{define-method-combination}.

\{\text{define-method-combination} c-type\}
\> Short Form. Define new \text{method-combination} \text{c-type} in a generic function using \text{c-type}, evaluate most specific \text{around} method supplying the values of the generic function. From within this method, \text{call-next-method} can call less specific \text{around} methods if there are any. If not, or if there are no \text{around} methods at all, return from the calling \text{call-next-method} or from the generic function, respectively, the values of \text{(operator (primary-method gen-arg") \*, gen-arg" being the arguments of the generic function. The primary-methods are ordered \{\text{most-specific-first} \}
\{\text{most-specific-last} \}
\> Using \text{c-type} as the qualifier in \text{defgeneric} makes the method primary.

\{\text{define-method-combination} c-type (ord-\lambda) \}
\> Long Form. Define new \text{method-combination} \text{c-type}. A call to a generic function using \text{c-type} will be equivalent to a call to the forms returned by \text{body"} with ord-\lambda bound to \text{c-type} (cf. \text{defgeneric}), with \text{symbol} bound to the generic function, with \text{method-combination-\lambda} bound to the arguments of the generic function, and with \text{groups} bound to lists of methods. An applicable method becomes a member of the left-most group whose \text{predicate} or \text{qualifiers} match. Methods can be called via \text{call-method}. Lambda lists (ord-\lambda) and (method-combination-\lambda) according to ord-\lambda on page 17, the latter enhanced by an optional \text{&whole} argument.

\{\text{call-method} \}
\> From within an effective method form, call method with the arguments of the generic function and with information about its \text{next-methods}; return its values.

11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 31.
### 13 Input/Output

#### 13.1 Predicates

- `(streamp stream)` → `T` if `stream` is of indicated type.
- `(pathnamep foo)` → `T` if `foo` is of indicated type.
- `(readablep foo)` → `T` if `foo` is of indicated type.
- `(input-stream-p stream)` → `T` if `stream` is for input, for output, interactive, or open, respectively.
- `(output-stream-p stream)` → `T` if `stream` is for input, for output, interactive, or open, respectively.
- `(interactive-stream-p stream)` → `T` if `stream` is for input, for output, interactive, or open, respectively.
- `(open-stream-p stream)` → `T` if `stream` is for input, for output, interactive, or open, respectively.
- `(pathname-match-p path wildcard)` → `T` if `path` matches wildcard.
- `(wild-pathname-p path ([:host] [device] [directory] [name] [type] :version NIL))` → `T` if indicated component in `path` is wildcard. (NIL indicates any component.)

#### 13.2 Reader

- `{y-or-n-p} (control arg)` → Ask user a question and return `T` or `NIL` depending on their answer. See page 36, `format`, for `control` and `args`.
- `{yes-or-no-p} (control arg)` → Ask user a question and return `T` or `NIL` depending on their answer. See page 36, `format`, for `control` and `args`.
- `(read)` → Return object read from string and zero-indexed position of next character.
- `(read-preserving-whitespace)` → Return printed representation of `object`.
- `(read-from-string string (eof-error [eof-error] [eof-value nil]))` → Read printed representation of `object`.
- `(read-delimited-list char)` → Continue reading until encountering `char`. Return list of objects read. Signal error if no `char` is found in `stream`.
- `(read-char)` → Return next character from `stream`.
- `(read-char-no-hang)` → Next character from `stream` or `NIL` if none is available.
- `(peek-char)` → Next, or if `mode` is `T`, next non-whitespace character, or if `mode` is a character, next instance of it, from `stream` without removing it there.
- `(unread-char character)` → Put last `read-char` character back into `stream`; return `NIL`.
- `(read-byte)` → Return next byte from `stream`.
- `(read-line)` → Read a line of text from `stream` and return `T` if line has been ended by end of file.

#### 13.3 Output

- `(repl)` → Return `T` if `repl` is of indicated type.
### 12 Types and Classes

For any class, there is always a corresponding type of the same name.

- `(typep footype [environment])` ➔ \(\top\) if footype is of type.
- `(subtypep type-a type-b [environment])` ➔ Return \(\top\) if type-a is a recognizable subtype of type-b, and NIL if the relationship could not be determined.
- `(the type form)` ➔ Declare values of form to be of type.
- `(coerce object type)` ➔ Coerce object into type.
- `(mtypecase footype (a-form)\(\top\))` ➔ Return values of the first a-form whose type is footype. If no type matches, return \(\top\).
- `(mctypecase footype (type form)\(\top\))` ➔ Return values of the first form whose type is footype. Signal non-correctable/correctable `type-error` if no type matches.
- `(typeof footype)` ➔ Type of footype.
- `(mcheck-type place type string)` ➔ Signal correctable `type-error` if place is not of type. Return NIL.
- `(stream-element-type stream)` ➔ Type of stream objects.
- `(array-element-type array)` ➔ Element type array can hold.
- `(upgraded-array-element-type type [environment])` ➔ Element type of most specialized array capable of holding elements of type.
- `(mdeftype footype (macro-λ)\{\declare decl\} form)` ➔ Define type footype which when referenced as `macro-λ` (or as `footype` if macro-λ doesn't contain any required parameters) applies expanded forms to args returning the new type. For `(macro-λ)` see page 23 but with default value of \(\star\) instead of `NIL`, forms are enclosed in an implicit `block` named `footype`.
- `(eq footype)` ➔ Specifier for a type comprising footype or footypes.
- `(satisfies predicate)` ➔ Type specifier for all objects satisfying predicate.
- `(mod n)` ➔ Type specifier for all non-negative integers < n.
- `(not type)` ➔ Complement of type.
- `(and type)` ➔ Type specifier for intersection of types.
- `(or type)` ➔ Type specifier for union of types.
- `(values type [optional type [rest other-args]])` ➔ Type specifier for multiple values.
- `*` ➔ As a type argument (cf. Figure 2): no restriction.
(print-newline [stream standard-output])

▷ Print a conditional newline if stream is a pretty printing stream. Return NIL.

*print-array* ▷ If T, print arrays readably.

*print-base* ▷ Radix for printing rationals, from 2 to 36.

*print-case* ▷ Print symbol names all uppercase (*upcase*), all lowercase (*downcase*), capitalized (*capitalize*).

*print-circle* ▷ If T, avoid indefinite recursion while printing circular structure.

*print-escape* ▷ If NIL, do not print escape characters and package prefixes.

*print-gensym* ▷ If T, print #: before uninterned symbols.

*print-length* ▷ If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.

*print-miser-width* ▷ If integer and greater than the width available for printing a substructure, switch to the more compact miser style.

*print-pretty* ▷ If T, print prettily.

*print-radix* ▷ If T, print rationals with a radix indicator.

*print-readably* ▷ If T, print readably or signal error *print-not-readable*.

*print-right-margin* ▷ Right margin width in ems while pretty-printing.

(*print-dispatch* type function [priority])

▷ Install entry comprising function of arguments stream and object to print; and priority as type into table. If function is NIL, remove type from table. Return NIL.

(*print-dispatch* [stream standard-output])

▷ Return highest priority function associated with type of foo and T if there was a matching type specifier in table.

(*copy-print-dispatch* [table standard-output])

▷ Return copy of table or, if table is NIL, initial value of *print-print-dispatch*.

*print-print-dispatch* ▷ Current pretty print dispatch table.

13.5 Format

(mformat control)

▷ Return function of stream and arg* applying *format* to stream, control, and arg* returning NIL or any excess args.

(*format* [stream control] control arg*)

▷ Output string control which may contain - directives possibly taking some args. Alternatively, control can be a function returned by *mformat* which is then applied to output-string and arg*. Output to out-string, out-stream or, if first argument is T, to *standard-output*. Return NIL. If first argument is NIL, return formatted output.

13.3 Character Syntax

multi-line-comment *#*

▷ Comments. There are stylistic conventions:

"..." ▷ Short title for a block of code.

"..." ▷ Description before a block of code.

"..." ▷ State of program or of following code.

; explanation ▷ Regarding line on which it appears.

(foo* [bar baz]) ▷ List of foos with the terminating cdr bar.

"foo" ▷ Begin and end of a string.

'(foo) ▷ (quote foo); foo unevaluated.

'((foo) [baz] [quux] [bing]) ▷ Backquote. quote foo and bing; evaluate baz and splice the lists baz and quux into their elements. When nested, outermost commas inside the innermost backquote expression belong to this backquote.

\c ▷ *character* +c*, the character c.

\Bn; \O; n.; \X; \R; r \Bn ▷ Integer of radix 2, 8, 10, 16, or r; 2 ≤ r ≤ 36.
Common Lisp Quick Reference

34

Common Lisp Quick Reference

\( n/d \)  
\( \{ m, n \} \cdot (\{ \text{short-float} \cdot \text{double-float} \cdot \text{long-float} \} \cdot z) \)

\( \text{m} \cdot \text{n} \cdot 10^z \)  
short-float, single-float, double-float, long-float, 
or the type from *read-default-float-format*.

#< (a b)  
(\text{complex} a b), the complex number \( a + bi \).

#*foo*  
(\text{function} foo); the function named foo.

#*n*  
n-dimensional array.

#*n* *b*  
Vector of some \( (\text{or} n) \) foos filled with last foo if necessary.

#*n* *b*  
Bit vector of some \( (\text{or} n) \) bs filled with last b if necessary.

#(type \{ slot value*})  
Structure of type.

#*string*  
A pathname.

#*foo*  
Uninterned symbol foo.

#*form*  
Read-time value of form.

#*read-eval*  
If NIL, a reader-error is signalled at #:.

#*integer#  
Give foo the label integer.

#*integer#  
Object labelled integer.

#<  
Have the reader signal reader-error.

#*feature when-feature*  
Means when-feature if feature is T; means unless-feature if feature is NIL. Feature is a symbol from *features*, or \((\text{and} \text{or}) \text{feature}^*)

:*features*  
List of symbols denoting implementation-dependent features.

\{c\}; \( \text{c} \)  
Treat arbitrary character(s) \( c \) as alphabetic preserving case.

13.4 Printer

\( \{\text{prin1}\} \)
\( \{\text{prin}\} \)
\( \{\text{pprint}\} \)
\( \{\text{princ}\} \)

\( \{\text{prin1-to-string} \text{foo} \} \)
\( \{\text{prin-to-string} \text{foo} \} \)
\( \{\text{pprint-to-string} \text{foo} \} \)
\( \{\text{princ return} \text{foo} \} \)

\( \{\text{print-object} \text{object} \text{stream} \} \)

\( \{\text{print-unreadable-object} \text{foo} \text{stream} \} \)

\( \{\text{terpri} \text{stream} \} \)

\( \{\text{fresh-line} \text{stream} \} \)

\( \{\text{write-char} \text{char} \text{stream} \} \)

\( \{\text{write-string} \text{string} \text{stream} \} \)

\( \{\text{write-line} \text{string} \text{stream} \} \)

\( \{\text{write-byte} \text{byte} \text{stream} \} \)

\( \{\text{write-sequence} \text{sequence} \text{stream} \} \)

\( \{\text{write} \text{foo} \text{to stream} \} \)

\( \{\text{write-to-string} \text{foo} \} \)

\( \{\text{write} \text{foo} \text{to stream} \} \)

\( \{\text{print-fill} \text{foo} \} \)

\( \{\text{print-tabular} \text{foo} \} \)

\( \{\text{print-linear} \text{foo} \} \)

\( \{\text{print-logical-block} \text{stream} \} \)

\( \{\text{print-pop} \} \)

\( \{\text{print-tab} \text{line} \text{stream} \} \)

\( \{\text{print-indent} \text{block} \text{stream} \} \)

\( \{\text{print-exit-if-list-exhausted} \} \)

\( (\text{declare decl}^*) \) \( \text{form}^* \)

\( \{\text{line} \text{line-relative} \text{section} \} \)

\( \{\text{c i} \text{stream} \} \)

\( \{\text{indent} \text{block} \text{stream} \} \)

\( \{\text{exit-if-list-exhausted} \} \)
13.7 Pathnames and Files

```
(defun make-pathname (host [device NIL] [directory NIL] [name NIL] [type NIL] [version NIL] [case :local common])
  (values host (merge-pathnames (merge-pathnames (merge-pathnames (merge-pathnames directory NIL) name) type) version) case)
```

The `make-pathname` function constructs a logical pathname. The translation for `host` can transform into a physical `pathname`. For `:case :local`, leave case of components unchanged. For `:case :common`, leave mixed-case components unchanged; convert all-uppercase components into local case; do the opposite with all-lowercase components.
Logical Block. Act like print-logical-block using body as format control string on the elements of the list argument or, with @, on the remaining arguments, which are extracted by print-pop. With ;, prefix and suffix default to ( and ). When closed by < -@ >, spaces in body are replaced with conditional newlines.

- [ [ ] [@ ]] < { [ [ ] [@ ] ] | [ [ ] [@ ] ] } body [ ; suffix @ ] >
  ▶ Logical Block. Act like print-logical-block using body as format control string on the elements of the list argument or, with @, on the remaining arguments, which are extracted by print-pop. With ;, prefix and suffix default to ( and ). When closed by < -@ >, spaces in body are replaced with conditional newlines.

- [ [ ] [@ ]] - [ [ ] [@ ] ]
  ▶ Indent. Set indentation to n relative to leftmost/to current position.

- [ [ ] [ ] @ ] \[ \]
  ▶ Tabulate. Move cursor forward to column number c + ki, k ≥ 0 being as small as possible. With @, calculate column numbers relative to the immediately enclosing section. With @, move to column number c0 + c + ki where c0 is the current position.

- [ [ ] [ ] @ ] + [ [ ] [ ] @ ]
  ▶ Go-To. Jump m arguments forward, or backward, or to argument n.

- [ limit | [ ] @ ] { text - }
  ▶ Iteration. Use text repeatedly, up to limit, as control string for the elements of the list argument or (with @) for the remaining arguments. With @ or @; list elements or remaining arguments should be lists of which a new one is used at each iteration step.

- [ x ; y [ -z ] ] ^
  ▶ Escape Upward. Leave immediately ≪ - >, ≪ < - >, ≪ { - } ≫, or the entire format operation. With one to three prefixes, act only if x = 0, x = y, or x ≤ y ≤ z, respectively.

- [ x | [ ] @ ] [ ( [ text - ]; text ) ; [ ; default ] - ]
  ▶ Conditional Expression. Use the zero-indexed argument (or rth if given) text as a format control subclause. With @, use the first text if the argument value is NIL, or the second text if it is T. With @, do nothing for an argument value of NIL. Use the only text and leave the argument to be read again if it is T.

- [ ? | @ ]
  ▶ Recursive Processing. Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original arguments.

- [ prefix ; prefix ] | [ ; package | : package | function / ]
  ▶ Call Function. Call all-upper-case package: function with the arguments stream, format-argument, colon-p, at-sign-p and prefixes for printing format-argument.

- [ [ ] @ ] W
  ▶ Write. Print argument of any type obeying every printer control variable. With @, pretty-print. With @, print without limits on length or depth.

- [ { } @ ]
  ▶ In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.

13.6 Streams

- Declaration
  ▶ direction : input ; output ; probe
  ▶ element-type : type ; \{ \}
  ▶ new-version : \{ \}
  ▶ error ; rename
  ▶ create : NIL
  ▶ if-exists : \{ \}
  ▶ if-does-not-exist : \{ \}
  ▶ external-format : format ; \{ \}

  ▶ Open file-stream to path.

- (make-concatenated-stream input-stream *)
- (make-broadcast-stream output-stream *)
- (make-two-way-stream input-stream-part output-stream-part)
- (make-echo-stream from-input-stream to-output-stream)
- (make-synonym-stream variable-bound-to-stream)
  ▶ Return stream of indicated type.

- (make-string-input-stream string [ start | end | nil ])
  ▶ Return a string-stream supplying the characters from string.

- (make-string-output-stream \{ element-type \{ \}
  ▶ Return a string-stream accepting characters (available via get-output-stream-string).

- (concatenated-stream-streams concatenated-stream)
- (broadcast-stream-streams broadcast-stream)
  ▶ Return list of streams. Concatenated-stream still has to read from/broadcast-stream is broadcasting to.

- (two-way-stream-input-stream two-way-stream)
- (two-way-stream-output-stream two-way-stream)
- (echo-stream-input-stream echo-stream)
  ▶ Return source stream or sink stream of two-way-stream/echo-stream, respectively.

- (synonym-stream-symbol synonym-stream)
  ▶ Return symbol of synonym-stream.

- (get-output-stream-string string-stream)
  ▶ Clear and return as a string characters on string-stream.

- (file-position stream [ start | end ] position)
  ▶ Return position within stream, or set it to position and return T on success.

- (file-string-length stream foo)
  ▶ Length foo would have in stream.

- (listen stream [ standard-input ])
  ▶ T if there is a character in input stream.

- (clear-input stream [ standard-input ])
  ▶ Clear input from stream, return NIL.

- (clear-output force-output [ [ ] stream ] standard-output)
  ▶ End output to stream and return NIL immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.
14.4 Standard Packages

common-lisp-cl

➤ Exports the defined names of Common Lisp except for those in the keyword package.

common-lisp-user-cl-user

➤ Current package after startup; uses package common-lisp.

keyword

➤ Contains symbols which are defined to be of type keyword.

15 Compiler

15.1 Predicates

(special-operator-p foo) ➤ T if foo is a special operator.

(compiled-function-p foo) ➤ T if foo is of type compiled-function.

15.2 Compilation

(compile NIL {definition}) ➤ Return compiled function or replace name’s function definition with the compiled function. Return T in case of warnings or errors, and NIL in case of warnings or errors excluding style-warnings.

(compile-file file \{other-args\}) ➤ Write compiled contents of file to out-path. Return true if successful; NIL if already loaded; return T if loaded; return NIL if already loaded; return T if successful.

(pathname file \{other-args\}) ➤ Pathname file writes to if invoked with the same arguments.

(load path \{other-args\}) ➤ Load source file or compiled file into Lisp environment. Return T if successful.

(compile-file-trivial name) ➤ Input file used by {compile-file-by}, \{load\}.

(compile \{print-file\} \{verbose\}) ➤ Defaults used by \{compile-file-by\} or \{load\}.

Common Lisp Quick Reference
14 Packages and Symbols

The Loop Facility provides additional means of symbol handling; see loop, page 21.

14.1 Predicates

(symbolp foo) \(\to\) T if foo is of indicated type.

(packagep foo) \(\to\) T if foo is of indicated type.

(keywordp foo) \(\to\) T if foo is of indicated type.

14.2 Packages

:bar | keyword:bar \(\to\) Keyword, evaluates to `bar.

package:symbol \(\to\) Exported symbol of package.

package::symbol \(\to\) Possibly unexported symbol of package.

(delete-package foo) \(\to\) Create or modify package foo with interned-symbols, symbols from used-packages, imported-symbols, and std-symbols. Add std-symbols to foo’s shadowing list.

(make-package foo) \(\to\) Create package foo.

(rename-package package new-name [new-nicknames]) \(\to\) Rename package. Return renamed package.

(in-package foo) \(\to\) Make package foo current.

(use-package other-packages [packages]) \(\to\) Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T.

(delete-package package) \(\to\) Delete package. Return T if successful.

*packages* \(\to\) Common Lisp user.

*all-packages* \(\to\) List of all packages.

*package-name* \(\to\) Name of package.

*package-nicknames* \(\to\) Nicknames of package.

(find-package name) \(\to\) Package with name (case-sensitive).

(find-all-symbols foo) \(\to\) List of symbols foo from all registered packages.

(find-symbol foo [package packages]) \(\to\) Intern or find, respectively, symbol foo in package. Second return value is one of `internal`, `external, or `inherited` (or NIL if `intern` has created a fresh symbol).

(unintern symbol [package packages]) \(\to\) Remove symbol from package, return T on success.

14.3 Symbols

A symbol has the attributes name, home package, property list, and optionally value (of global constant or variable name) and function (function, macro, or special operator name).

(make-symbol name) \(\to\) Make fresh, uninterned symbol name.

(gensym [n]) \(\to\) Return fresh, uninterned symbol #\$an with n from *gensym-counters*. Increment *gensym-counters*.

(gentemp [prefix [package packages]]) \(\to\) Intern fresh symbol in package. Deprecated.

(copy-symbol symbol [props]) \(\to\) Return unshared copy of symbol. If props is NIL, give copy the same value, function and property list.

(symbol-name symbol) \(\to\) Name or package, respectively, of symbol.

(symbol-package symbol) \(\to\) Property list, value, or function, respectively, of symbol.

(symbol-value symbol) \(\to\)
(eval-when (compile-toplevel load) (execute-eval) form)

▷ Return values of forms if eval-when is in the top-level of a file being compiled, in the top-level of a compiled file being loaded, or anywhere. Return NIL if forms are not evaluated. (compile, load and eval deprecated.)

(locally (declare (eval) form))

▷ Evaluate forms in a lexical environment with declarations decl in effect. Return values of forms.

(with-compilation-unit ((override book)) form)

▷ Return values of forms. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of forms.

(load-time-value form (read-onlyform))

▷ Evaluate form at compile time and treat its value as literal at run time.

(quote foo)

▷ Return unevaled foo.

(make-load-form saving-slots foo (environment environment))

▷ Return a creation form with an initialized initialform that when evaluation construct an object equivalent to foo with slots initialized with the corresponding values from foo.

(macro-function symbol) (environment environment)

▷ Return specified macro function, or compiler macro function, respectively, if any. Return NIL otherwise. setfable.

(macro-car form)

▷ Return value of value of form evaluated in global environment.

15.3 REPL and Debugging

(\+t \+t \+t \+t \+t \+t)

▷ Last, penultimate, or antepenultimate form evaluated in the REPL, or their respective primary value, or a list of their respective values.

(apropos string (package))

▷ Print interned symbols containing string.

(apropos-list string (package))

▷ List of interned symbols containing string.

(diddle [path])

▷ Save a record of interactive session to file at path. Without path, close that file.

(ed [file-or-function])

▷ Invoke editor if possible.

(macropexpand-1 form (environment))

▷ Return macro expansion, once or entirely, respectively, of form and if form was a macro form. Return form and NIL otherwise.

(macropexpand form (environment))

▷ Function of arguments expansion function, macro form, and environment called by macropexpand-1 to generate macro expansions.
15.4 Declarations

(declare decl*)
- Globally make declaration(s) decl. decl can be: declaration, type, ftype, inline, notinline, optimize, or special. See below.

(declare decl)
- Inside certain forms, locally make declarations decl*. decl can be: dynamic-extent, type, ftype, ignorable, ignore, inline, notinline, optimize, or special. See below.

(declaration foo*)
- Make foos names of declarations.

(dynamic-extent variable* (function function)*)
- Declare lifetime of variables and/or functions to end when control leaves enclosing block.

((type type variable*)
(ftype type function*)
- Declare variables or functions to be of type.

(ignorable (var
(ignore (function function))
- Suppress warnings about used/unused bindings.

(inline function*)
(notinline function*)
- Tell compiler to integrate/not to integrate, respectively, called functions into the calling routine.

(optimize (compilation-speed (compilation-speed
  debug (debug
    space (space
      speed (speed
    )
  )
  )
)
- Tell compiler how to optimize. n = 0 means unimportant, n = 1 is neutral, n = 3 means important.

(special var*)
- Declare vars to be dynamic.

16 External Environment

(get-internal-real-time)
- Current time, or computing time, respectively, in clock ticks.

(get-internal-units-per-second)
- Number of clock ticks per second.

(encode-universal-time sec hour date month year [zone])

(get-universal-time)
- Seconds from 1900-01-01, 00:00, ignoring leap seconds.

(decode-universal-time universal-time [time-zone])

(get-decoded-time)
- Return second, minute, hour, date, month, year, day, daylight-p, and zone.

(short-site-name)

(long-site-name)
- String representing physical location of computer.

(lisp-implementation)

(machine)
- Name or version of implementation, operating system, or hardware, respectively.

(machine-instance)
- Computer name.